John von Neumann Institute for Computing

A New Algorithm for Join Processing with the Internet Transfer Delays

K. Imasaki, S. Dandamudi

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 499-506, 2006.

http://www.fz-juelich.de/nic-series/volume33

A New Algorithm for Join Processing with the Internet Transfer Delays

Kenji Imasaki [a], Sivarama Dandamudi[a]

[a]School of Computer Science, Carleton University 1125 Colonel By Drive, Ottawa, Canada

This paper focuses on cluster-based parallel database systems in which only one of the nodes has the database and the other nodes, which have no initial data, are used for parallel query processing. In such a system, the load of each node changes dynamically depending on the activities of the local users. In addition, in database query processing, data skew exists. With the increasing Internet connectivity, it also becomes necessary to query databases spread around the world. In this scenario, the system can experience arrival delays and/or the transfer rate variations while receiving the join input relations. This paper investigates join processing algorithms under these conditions and proposes a new join algorithm called Symmetric Chunking Hash Join (SCHJ) that divides the hash buckets into chunks and uses them for load balancing. The SCHJ is compared with two incremental hash mapping algorithms. The experimental results conducted on a Linux cluster show that the SCHJ algorithm is the best among these algorithms.

1. Introduction

With the availability of Giga-hertz processors, Giga-byte memory, and Giga-bit bandwidth communication networks, huge parallel processing power from parallel computers can be used for various types of scientific computing. However, this power does not come for free as parallel systems are very expensive. Also, with the fast pace of technological advances, constituent components of the parallel computers quickly become obsolete. *Cluster systems* have been introduced as an alternative to such parallel systems. Database query processing can also benefit from parallel execution on such cluster systems. The query processing is managed by a Parallel Database Management System (PDBMS).

With the advent of cluster computing environments, parallel query processing on a cluster system has been proposed as an alternative to parallel database systems. In general, there are two approaches to implementing a PDBMS on a cluster system. One is the same as a traditional PDBMS: the data are de-clustered and a query is executed in parallel. Most of the recent commercial PDBMSs use this approach. The other approach is to use an existing dedicated DBMS and processing nodes (PNs) in clusters for parallel processing to take the query load off the DBMS [3,7]. We call this system a cluster-based PDBMS (*cPDBMS*) to distinguish the two approaches. This paper focuses on query processing in a cPDBMS.

When processing a query, choosing an efficient parallel query processing algorithm is important. Query processing can be improved by exploiting intra-operator (single-join), inter-operator (multiple-join), and inter-query (multiple query) parallelism [1].

Among these three types of parallelism, the single-join operation has attracted a lot of attention, since it is the most expensive operation in query processing. Hash join algorithms are clearly superior than other algorithms for the single-join operation [11]. However, hash join-based algorithms suffer from various kinds of skew [9]. Thus, the choice of load balancing/sharing algorithms becomes important since the slowest PN which has the heaviest data skew dictates the performance of the overall system.

Many researchers have proposed load sharing/balancing algorithms for the hash join algorithm [2,7,11]. Also, Hua et al. [5] compared the performance of the following load balancing algorithms on a shared-nothing parallel computer: (1) no load balancing, (2) conventional bin-packing, (3)

sampling, and (4) incremental methods. They concluded that the sampling method is the best.

These load balancing/sharing algorithms for PDBMSs on parallel computers do not work for cPDBMSs for the following reasons.

Firstly, these algorithms only deal with PNs with pre-partitioned data. However, PNs in clusters are dynamically determined and usually do not have any of the data needed for join processing. The data should be sent from DBMSs to a PN prior to the join processing. This phase is not considered in these algorithms. This situation can be seen as the extreme case of tuple placement skew [9]. In tuple placement skew, some PNs read more tuples while others wait for them to finish reading the whole relation. Secondly, these algorithms do not consider the effect of non-query background load. Even with the adaptive approach, it is difficult to know how much work a PN should transfer to another. Besides, it is not clear whether the transfer is effective or not in the case of clusters in which the load on each PN changes very frequently. In addition, no algorithm considers the effect of the combination of background load and data skew. Lastly, input data arrival delay and data transfer rate fluctuation are not considered. This is very important, especially in the case of data integration.

Therefore, a new load balancing/sharing algorithm is needed to improve the performance of join processing on clusters. This paper proposes a new load balancing/sharing algorithm for cPDBMSs. This algorithm, called Symmetric Chunking Hash Join (SCHJ), divides the hash buckets into chunks and uses them for load balancing. Also, the algorithm is based on the symmetric join algorithm, which does not distinguish between the two input relations.

2. Symmetric Join Algorithms

The symmetric hash join algorithms were proposed by Wilchut et al. [14]. Also, recently developed algorithms for data integration systems (i.e., XJoin [13]) are based on symmetric single-join algorithms. We combine symmetric hash join algorithms with the ChunkHJ algorithm proposed in [7]. In this section, we first explain the environment. Then, the load balancing/sharing algorithms are described. Next, experimental environments, including the Internet transfer delay model, are discussed.

2.1. Single-Join Processing Environment

This subsection presents the environments for symmetric single-join processing. In this environments, data is coming from remote sites to the local cluster , which consists of several processing nodes with single or dual CPUs. The local cluster is used for parallel query processing.

We developed a system to simulate query processing in the local cluster in this model. A description of each component of this system is shown in Table 1. All components are implemented by Java threads, which run concurrently on PNs. The main focus of this paper is on the JoinManager, JoinExecutors, and the Database. The JoinManager reads data from a Database and coordinates load balancing/sharing of several JoinExecutors.

With the same idea as ChunkHJ, a lot of hash bucket chunks are created using threshold values for load balancing/sharing purpose. In order to ensure the correctness of the join execution, a Join State Matrix (JSM) resides on the JoinManager is used. Each entry in the JSM represents the join status of matching pairs for each bucket.

2.2. Load Sharing/Balancing Techniques for Symmetric Hash Join Algorithms

We designed and implemented several load balancing/sharing algorithms for symmetric hash joins. The main focus is to determine hash mapping from hashId to JoinExecutorId. It is stored in *hashMappingTable* for load sharing/balancing. We developed three algorithms to decide the hash mapping. The following subsections explain these algorithms in detail. The functions used in the pseudocode description in the following subsections are summarized in Table 2. The *recv*, *send*, and *broadcast* functions are based on the MPI functions.

Class	Description
JoinMgr	makes load-balancing decision (JM)
JoinExec	executes local joins (JE)
TranExec	transfers chunks
DBRead	reads rel. and puts into buffer (DR)
DBMgr	accepts a read request and invokes DRs
Backgrd	simulates non-query process by busy looping
HashGen	uses function to generate buckets (HG)
ChunkStr	stores the chunk and deals with I/Os

Table 1
Major components descriptions.

Name	Description
recv	receives a *msg* from the master or a slave (*srcId*) with message *tag*
send	sends a *msg* with a message *tag* to the master or a slave (*destId*)
broad	broadcasts a *msg* to all slaves
apHash	applies a hash function and creates *n* hash buckets
read	reads from DBMS within the range
execLJ	executes the local join algorithm and stores results in the result buffer;

Table 2
Functions used in the pseudocode description.

The pseudocode for JoinManager is shown in Algorithm A.1. The pseudocode for HashGenerator is also shown in Algorithm A.2. *expandJSM* is used to expand the JSM entry according to the argument *info* (a pair of hashId and chunkId) sent from HashGenerator. It also fills the expanded matrix entries with "R"(Ready) entry. *FindJE* is the sender-initiated part of the algorithm that tries to find an idle JoinExecutor with the maximum number of matching pairs for this bucket. Each algorithm has a different version of *findJE* as we will explain later. *FindBucket* is used to find a suitable bucket pair with an idle JoinExecutor. For the hash mapping-based algorithm (non-SCHJ algorithms), hash mapping is used to find a suitable bucket pair. For SCHJ, the algorithm described in Section 2.3 is used. Pseudocode for JoinExecutor is shown in Algorithm A.3. *RSTransfer* is used to transfer the bucket among DB and JoinExecutor for load balancing/sharing purpose.

2.3. Symmetric Chunking Hash Join

This algorithm (SCHJ) is based on ChunkHJ [7]. Thus, hash mapping is not used. The pseudocode of *findJoinExecutor(h,cId,chunkSize)* of *SCHJ* is shown in Algorithm A.4. In this algorithm, *findSlave* [7] is used. It finds an idle slave[1] with the other matching bucket. However, the buckets which have "R" entries in JSM are considered.

Every time a JoinExecutor finishes the job, the JoinManager invokes *findBucket* (Algorithm A.5) after receiving a job request message from a JoinExecutor. *findBucket* selects a chunk pair using *findBucket(LT,chunkSize)*. Then the *findBucket(HT,chunkSize)* algorithm [7] finds a bucket in which the number of tuples is greater than the value specified in the parameter.

After a bucket chunk pair is selected, chunk transfer is done using *RSTransfer*. If at least one of the chunk pairs is not present on a JoinExecutor, then it is sent from the source node to the destination node by TransferExecutor on the source node. The source node is determined randomly.

This algorithm may perform well in the case that background load and/or data skew exist since load balancing/sharing is done in a dynamic and incremental way. However, too many transfers may cause performance degradation.

2.4. Greedy Incremental Hash Mapping and JSM-based Incremental Hash Mapping

Greedy Incremental Hash Mapping (GIHM) and JSM-based Incremental Hash Mapping (JIHM) try to deal with the Internet transfer delay that occurs in the data integration systems. The basic

[1]The terms "slave" and "JoinExecutor" are used interchangeably.

idea of these algorithms is to delay JoinExecutors to work on a bucket until there is enough work available in the bucket. The unit of work measurement is the number of tuples (GIHM) or the number of join-ready entries in JSM (JIHM).

GIHM uses a greedy algorithm and incrementally determines the hash mapping according to the arrivals of hash chunks. The pseudocode of *findJE(h)* (GIHM) is shown in Algorithm A.6. *FindBucket* uses hash mapping (omitted here). This algorithm is greedy in the sense that the JoinManager looks for an idle JoinExecutor and assigns the JoinExecutor to the hashId (hash mapping) immediately. When there are several JoinExecutors, one of them is chosen randomly as a target JoinExecutor. After the target JoinExecutor is selected, the JoinExecutor receives the hash chunk from TransferExecutor in the same way as *SCHJ* and starts the join.

In JIHM, JoinManager waits for the hash mapping assignment until the number of JSM-ready entries reaches a pre-defined number.

JIHM and GIHM may perform well when the arrival relation is delayed due to dynamic characteristics of the Internet transfer.

3. Experimental Environment

The LINUX cluster that we used in our experiments consists of 4 dual CPU nodes (Xeon 2.4 GHz and 1 GB main memory) and 7 single CPU nodes (P4 2.4 GHz and 1 GB main memory). Each node runs the Red Hat 8.0 Linux.

MPI (Message Passing Interface) [10] is now the de facto standard for programming languages for parallel processing. We use mpich 1.2.5.2 since it has thread-safe architecture and our simulation uses several threads running concurrently on PNs.

We use mpiJava 1.2.5 [4] with JDK 1.4.1 to combine the advantage of MPI and Java. MpiJava is an object-oriented Java interface to the standard MPI. MpiJava itself does not assume any special extensions to the Java language. It is portable to any platform that provides compatible Java-development and native MPI environments. We did not use pure Java parallel processing such as RMI for performance reasons.

We set memory size for a component to 24MB and node allocation sequence to interleave (single CPU, dual CPUs, single, ...) and background loop period to 1 second.

The experimental database used in the experiments consists of 1 million tuples. We created 10 relations with different random seeds for each parameter setting. In this experiments, scalar skew model [7]. In scalar skew mode, for a relation of size $|R|$, in each attribute the value 1 appears in a fixed number of tuples, while the remaining tuples contain values uniformly distributed between 2 and $|R|$.

In order to model data transfer delay over the Internet, we first measured the transfer time of different message size by UNIX ping command [12] 100 times at 3 different times of the day from our office in Ottawa to several locations spanning a range of distances from University of Toronto, University of California Los Angles (UCLA), to University of Tokyo.

Figure 1 and Table 2 show ping transfer time and approximate functions for each location.

We decided to use hypo-exponential distribution to simulate data transfer rate fluctuation over the Internet since coefficient of variation (CV) (= standard deviation/average time) is below 1. As a result, we use the following model to get the transfer times (*ActualTransferTime*) of data as a function of *size*:

- MeanTransferTime = a**size* + b (column 4 of Table 2)
- ActualTransferTime = Hypo_exponential (*MeanTransferTime,dev*) where *dev* is its standard deviation

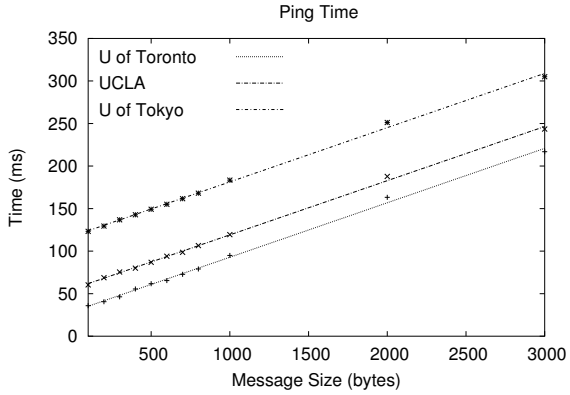

Figure 1. Ping transfer time: trend line is shown in column 3 of Table 2.

Location	Approx. Transfer Func. (trend lines in Fig. 1)
Univ. of Toronto	t=0.064×size+28.839
UCLA	t=0.0637×size+55.416
Univ. of Tokyo	t=0.0638×size+117.57

Figure 2. Approximate transfer functions.

We believe that this model is the first step to model the Internet transfer delay accurately. The statistical analysis of this model is our future work.

We insert a sleeping function after reading the relation and before applying the hash function with a duration that corresponds to the above model.

In the experiments, it is assumed that one relation resides on the local area network and the other relation resides in another location (either at Toronto, UCLA, or Tokyo). The reason for this decision is that if both of them reside on remote locations, then it is better to execute the join on one of the remote locations.

4. Experimental Results

The experimental results obtained by executing the symmetric hash join algorithms and their load balancing/sharing algorithms described in Section 2.2 are presented in this section under the data skew, background load, and the Internet transfer delay conditions. The execution is repeated until 99% confidence interval is obtained. For GIHM, we use 5, 10, and 20 as its threshold values. The complete results can be found in [6].

4.1. Performance with the Internet Transfer Delay and Data Skew

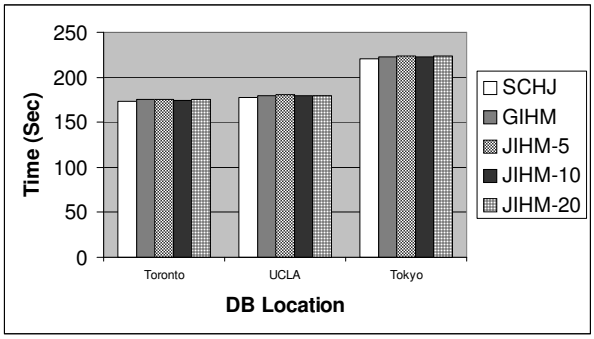

Figure 3. Performance of algorithms with scalar data skew factor of 20,000.

Figures 3 show the performance of the algorithms when there is no background load and the scalar skew factor is 20000 which changing DB location. SCHJ is marginally better than the other algorithms for all DB locations and all the degrees of skew.

In the figure, the delay caused by the Internet transfer delay from one location to another location is 4% (from Toronto to UCLA) and 25% (from UCLA to Tokyo). Table 1 shows the delay is 13% (from Toronto to UCLA) and 26% (from UCLA to Tokyo). Thus, when the delay is small the algorithms can absorb the delay. However, as the distance becomes long, the algorithms are affected by the delay.

Another interesting point is that the skew effects on the performance are not as large as the case without the Internet transfer delay [6]. This is because JoinExecutors can work on the join processing on skewed bucket while waiting for the relations to arrive as long as there is no background load on them. If there is a background load, it delays the join execution as we will see in Section 4.3.

4.2. Performance with the Internet Transfer Delay and Background Load

Figure 4 shows the performance of the algorithms as the function of background load when the DB location is Tokyo. This plot shows that the effect of the background load is small on SCHJ compared to the other algorithms because of its adaptive load balancing/sharing mechanism. Thus, the higher background load, the higher the improvement. SCHJ improvements over GIHM are 2% and 5% when the background load is 3 and 6 processes, respectively.

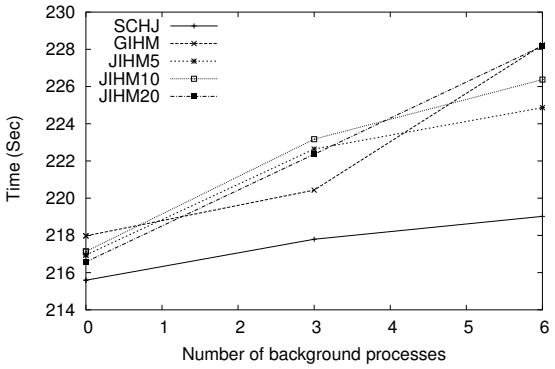

Figure 4. Performance of algorithms with the DB location as Tokyo and no skew.

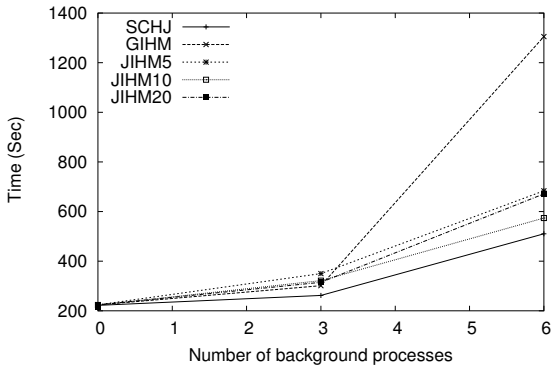

Figure 5. Performance of the algorithms with the DB location as Tokyo and the skew factor of 20,000.

4.3. Performance with the Internet Transfer Delay, Data Skew and Background Load

Figure 5 shows the performance of the algorithms as a function of the number of background load processes when the skew factor is 20000 and the DB location is Tokyo. The figure shows that SCHJ is the least affected by the background load. Thus, the higher the background load, the more SCHJ improves compared to other algorithms, which are affected by the change in background load.

Performance of SCHJ is 11% better than JIHM10, which shows the effectiveness of SCHJ. JIHM10 is better than the other JIHMs and GIHM. When the skew factor is high, the effect of the background load is severe on GIHM. GIHM is good for moderate skew case and low background load. In case of high skew and high background load, it does not perform well because of its greedy algorithm which results in poor hash mapping decision. On the other hand, JIHM waits for more data to arrive before it makes a decision. Among the various JIHM algorithms, smaller number (5 or 10) of ready JSM entries is better in these cases. If it is 20, it waits too long and keeps JEs idle long.

5. Conclusions

In this paper, we first proposed Symmetric Chunking Hash Join (SCHJ). We also proposed Greedy Incremental Hash Mapping (GIHM) and JSM-based Incremental Hash Mapping (JIHM) mainly for the Internet transfer delay case. They are compared with SCHJ for the Internet transfer delay model. In the model, one of the relations resides locally and the other resides at a remote location that has a dynamic transfer delay depending on the geographical distance. The reason for assuming a local relation is that if both of them reside on remote locations, then it is better to execute the join on one of the remote locations.

We draw the following conclusions: (1) With only the Internet transfer delay or with data skew (scalar skew), SCHJ is marginally better than the other algorithms. In this case, data skew can be absorbed by the arrival delay if there is no background load for all algorithms. (2) The greater the background load, the better the SCHJ performance because of the load balancing mechanism. (3) The improvement of SCHJ becomes greater with the increasing data skew or the background load. (4) When there is modest skew and background load, GIHM is better than the JIHM algorithms but worse than SCHJ (5) When there is extreme skew and background load, JIHM is better than the GIHM algorithm but worse than SCHJ.

References

[1] David J. DeWitt and Jim Gray. Parallel Database Systems: The Future of High-Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[2] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and Srinivasan Seshadri. Practical Skew Handling in Parallel Joins. In *The 18th VLDB*, pages 27–40, August 1992.

[3] Matthiew Exbrayat and Lionel Brunie. A PC-NOW Based Parallel Extension for a Sequential DBMS. In *PC-NOW*, pages 91–100, May 2000.

[4] HP Java Project. mpiJava Home Page. available at http://www.javagrande.com/mpiJava.html.

[5] Kien A. Hua and Wallapak Tavanapong. Performance of Load Balancing Techniques for Join Operations in Shared-nothing Database Management Systems. *Journal of Parallel and Distributed Computing*, 56(1):17–46, January 1999.

[6] Kenji Imasaki. *Parallel Query Processing on a Cluster-based Database System*. PhD thesis, School of Computer Science, Carleton University, September 2004.

[7] Kenji Imasaki and Sivarama Dandamudi. An Adaptive Hash Join Algorithm on a Network of Workstations. In *IPDPS*, April 2002.

[8] Holger Märtens. Skew-Insensitive Join Processing in Shared-Disk Database Systems. In *International Workshop on Issues and Applications of Database Technology*, pages 17–24, July 1998.

[9] Holger Märtens. A Classification of Skew Effects in Parallel Database Systems. In *European Conference on Parallel Computing (EURO-PAR)*, pages 291–300, Manchester, United Kingdom, August 2001.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, Knoxville, Tennessee, June 1995.

[11] Donovan A. Schneider and David J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-nothing Multiprocessor Environment. *SIGMOD Record*, 18(2):110–121, June 1989.

[12] W. Richard Stevens. *UNIX Network Programming; Volumne I, Networking APIs: Sockets and XTI, Second Edition*. Prentice Hall PTR, 1998.

[13] Tolga Urhan and Michael J. Franklin. XJoin: Getting Fast Answers From Slow and Bursty Networks. Technical Report CS-TR-3994, University of Maryland, College Park, February 1999.

[14] Annita N. Wilschut, Peter M. G. Apers, and Jan Flokstra. Parallel query execution in PRISMA/DB. In *Proceedings of Parallel Database Systems*, pages 424–433, September 1991.

A. Algorithms

Algorithm A.1: JOINONJOINMANAGER(cS)

{Input: cS for chunk size; Output: none}
$send(HashGenerator$, "RelationRead", "R")
$send(HashGenerator$, "RelationRead", "S")
repeat
  $recv(source, tag, info)$ {info:hashId/chunkId}
  if (*tag* is "JSMUpdate") {from HG}
    then $expandJSM(info)$
      $JEx \leftarrow findJE(info, cS)$;
        {different for each algorithm}
      $send(src$, "JSMUpdateReply", $JEx)$
  else if (tag is "JobRequest"){from JE}
    then change corresponding completed JSM entries to "F"
      $chunkPair \leftarrow findBucket(cS)$
      $RSTransfer(chunkPair)$
until ((finished reading relations(R and S)) and (JSM entries are all "F"))
$broadcast$("ProcessEnd", $null$){to all JEs/HGs}

Algorithm A.2: JOINONHG(X, pS, cS)

{applies hash function on tuples of X}
{Input: relation X;
pS for partition size for relation X;
cS for chunk size; Output: none}
repeat
  $recv(JoinManager$, "RelationRead", $X)$
    {Read relation X by Database and DatabaseReader}
  $nPartitions \leftarrow |X|/pS$
  for $i \leftarrow 0$ **to** $nPartitions$
    do $X^i \leftarrow read(X, i * pS, (i + 1) * pS - 1)$
      {actual reading is done by DatabaseManager}
      $X^i_{ALL} \leftarrow applyHash(X^i, cS)$
until $recv(JoinManager$, "ProcessEnd", $null)$

Algorithm A.3: JOINONJE()

{Symmetric Hash Join on a JoinExecutor}
{Input: none; Output: none}
repeat
  $recv(TransferExecutor$, "Relation", $X_h)$
  if(X is R)
    then $execLJ(X_h, S_h)$
    else $execLJ(R_h, X_h)$
  $send(JoinManager$, "JobRequest", $null)$
until $recv(JoinManager$, "ProcessEnd", $null)$

Algorithm A.4: FINDJE$_{SCHJ}(h, cId, cSize)$

{find a JoinExecutor(JEx) using Chunk HJ}
{Input: h for hashValue; cId for chunkId; cSize;
Output: destination JE index}
$SLx \leftarrow findSlave$("LT", h) {shown in [7]};
{LT : SLx should have other paring bucket.}
if (SLx is null)
  then $SLx \leftarrow findSlave$("HT", h)
{HT : SLx does not need the other pairing bucket.}
return (SLx)

Algorithm A.5: FINDBUCKET($cSize$)

{Find suitable bucket chunk pair from JSM Entry}
{Input: chunkSize;
Output:bucketPair(hashId,chunkId1,chunkId2)}
$bucketPair \leftarrow findBucket$("LT", $cSize$)
    {find local hash bucket but use JSM [7]}
if (bucketPair is null)
  then $bucketPair \leftarrow findBucket$("HT", $cSize$)
return $(bucketPair)$

Algorithm A.6: FINDJE$_{GIHM}(h)$

{Decide destination JoinExecutor}
{Input: h hashValue;
Output: destination JE index
if (hashMappingTable.containsKey(h))
  then return $(hashMappingTable.get(h))$
  else $x \leftarrow$ random selection from idle JE list
    $hashMappingTable.insert(h, x)$
    return (x)